En-co.de: A web service for augmenting physical objects with an active digital presence.

APPROVED BY

SUPERVISING COMMITTEE:

Adnan Aziz, Supervisor

Aaron Knoll

En-co.de: A web service for augmenting physical objects with an active digital presence.

by

Brandt Michael Westing, B.S.E.E.

REPORT

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN ENGINEERING

THE UNIVERSITY OF TEXAS AT AUSTIN

August 2013

I would like to dedicate this report to my mom, Delaina Westing. From kindergarten to graduate school, you always pushed me to never accept anything but my best. I wouldn't be where I am today, without you.

Acknowledgments

I would like to give a special acknowledgment to Adnan Aziz, my academic supervisor for graduate school. Adnan introduced me to the wide world of web development, inspired me with his enthusiasm for algorithms and software engineering, and gave me a set of tools that helped me achieve the position I currently have in the software industry.

En-co.de: A web service for augmenting physical objects with an active digital presence.

Brandt Michael Westing, M.S.E.
The University of Texas at Austin, 2013

Supervisor: Adnan Aziz

It is now possible for physical objects to have a dynamic digital presence via active identification codes that can be scanned via ubiquitous devices such as smart phones or tablets. *En-co.de* is a web service for the generation, storing, retrieval, and augmentation of metadata associated with physical objects. The service is built upon the concept that metadata associated with an object can be retrieved through a Quick Response (QR) coded URL. *En-co.de* serves to link a physical entity to a digital archive of information and provides services such as geolocation and SMS text alerts when an object's identifier, or tag, is scanned. I provide an analysis of QR code qualitative characteristics, the architecture for the *en-co.de* web service, a scalability study of the *en-co.de* architecture, and the results of the completed service in production in this report. In addition, the analysis is complemented with an evaluation of comparable identification schemes and web services.

Table of Contents

List of Tables

List of Figures

Chapter 1

Giving a Digital Presence to Physical Objects

Today it is possible for nearly all physical objects to have a digital presence. Augmenting physical objects with digital identities—that is, associating a physical object with an active digital data archive available on the Internet—enables interaction with objects in ways that can increase their usability, longevity, and information capacity beyond their existence in the physical world. These augmented capabilities can be exploited to track an object's position, understand its qualities, catalog or sort collections of objects, and build relationships between objects.

Today, however, not all objects have digital identities due to reasons including practicality, cost, and public awareness. *En-co.de* is a web service for granting an inexpensive and practical digital identity to any physical object. Through a guided web application, *en-co.de* allows users to create physical tags in the form of Quick Response (QR) codes, a form of two-dimensional bar code. The service associates each tag with a cloud-hosted archive that can be modified by the user. These tags can be scanned by others using a smart phone or tablet computing device, whereby the object's meta data can be retrieved or its cloud-hosted archive updated on behalf of the owner. *En-*

co.de makes augmenting physical objects practical through an easy-to-use web interface; affordable as tags are printable from any standard home printer; and builds upon an accepted standard—the QR code—to associate with a highly visible and publicly acceptable digital encoding technology.

1.1 User Stories

The following user stories motivate the *en-co.de* web service and explain its purpose in realistic situations.

1.1.1 The Library

Art Vandalay is an architect with a passion for reading. He has a large library of books, many of which he often lends out to friends, and some of which he are never returned or are lost. Art needs a way to associate the books to him, but writing his name and address on their inside cover is not ideal for Art: his address changes frequently. Art needs a way to associate his books to him, track their position if lost, and maintain a digital catalog of his physical library.

Art uses *en-co.de* to provide a digital presence for each of his books. After signing up for an *en-co.de* account, Art is able to batch print hundreds of tags for his books. Upon attaching the tags to his books, he scans each tag with his smart phone for a first time to initialize their digital data: he inserts a name and description of the book using his smart phone. This data is saved to the *en-co.de* database.

When Art now lends out a book, he can be sure that the tag associated with the book can always maintain up-to-date information: he simply changes his address on the *en-co.de* website, and any time a book with his tag is scanned, his updated address will show to the borrower. Art has also enabled location tracking on his tags through *en-co.de*: whenever a borrower, or someone who finds his lost book scans the tag, the location of the book at the time of the scan is reflected by the *en-co.de* service and Art is notified with an email and SMS text message. Art now has more confidence when lending his books.

1.1.2 The Veterinarian

Veterinarian Sarah Jane is opening a new pet clinic in Austin, Texas and is looking for ways to set her business apart from competitors. When a customer brings a new pet by the clinic, Sarah would like to give the owner a *Getting Started Packet* that contains items such as a dog collar and instructions for how to best care for their new pet. Sarah also includes an *en-co.de* tag for new customers that they can attach on their new dog collar. If their animal is lost in the future and someone scans the *en-co.de* tag on the animal's collar, the owner will be updated by the web service with the location and time the tag was scanned. The friendly stranger who finds the animal can easily inform the owner they found the pet using the meta data retrieved when scanning the tag, which can include the owner's phone number and email address.

1.2 Project Development Overview

The development of the *en-co.de* web service focused on three key objectives: identifying the format and properties of the physical tag that would be attached to objects to give them a digital presence, the development of a web-service back-end for storing, retrieving, and cataloging meta data related to the physical tags, and the development of a user-friendly front-end interface to the web service.

Identifying the properties of the physical tags and choosing a representative technology centered around finding a physical form that could encode a small amount of information while simultaneously being inexpensive and easy to obtain. Beyond these qualities, the information encoded on the tags must be capable of being decoding by devices that an average person will possess. In Section 4 the quantitative and qualitative properties of the chosen technology, the QR code, is explained in detail as it pertains to *en-co.de*.

The development of the web service back-end involved the investigation of technologies that offer persistent data storage and fast, inexpensive data access. Upon identifying the Google App Engine as a platform, the back-end resources were developed using established software engineering principles that focused on scalability. More details of the back-end development and platform can be found in Section 2.2.

The creation a user-friendly front-end interface to the *en-co.de* service was the third major development of the project. Work focused on identi-

fying modern web development practices and implementing a web site that is scalable and responsive to many types of browsing devices. An in-depth explanation of the front-end interface development can be found in Section 2.3.

En-co.de is in many aspects inspired from previous work I completed as a final project in the Advanced Programming Tools course as part of the University of Texas Masters in Software Engineering degree coursework. The final project resulted in *iDogTags*, an early prototype of *en-co.de*. *iDogTags* was a website for generating QR codes for use in dog tags or dog collars, and was designed as a product for pet owners who wanted to be able to more easily find their lost animals. While *iDogTags* did not support text message alerts, accurate location tracking, or a clean user interface, it was built on a technology stack similar to *en-co.de*. *En-co.de* was developed with software testing through unit tests as a major component, and is far more comprehensive in capabilities. *En-co.de* features a clean and responsive user interface and polished user experience when compared to *iDogTags*. Furthermore, *En-co.de* is composed of 2.5x the number of lines of code when compared to *iDogTags*.

1.3 Summary of Contributions

En-co.de is a web service for augmenting physical objects with an active digital presence. The web service provides facilities for describing objects and creating physical tags in the form of QR codes that can be attached to objects. These QR codes encode a unique identifier and URL to every object such that

the object carries with it a link to its digital existence. *En-co.de* provides a web interface for managing meta data associated with physical objects, tracking their position, and alerting the owner of the object that a tag was scanned through SMS text messages and email. In addition, *en-co.de* provides a REST API upon which others can build applications using the service.

This report describes the *en-co.de* web service and its architecture. In addition to describing how *en-co.de* works, the report explains the choice of QR tags as the physical mechanism by which augmentation works and the motivating factors behind choosing QR tags as this mechanism. The architecture and design of *en-co.de* is described in addition to the cost of the service and software engineering metrics that together give information about the scope of the *en-co.de* project. The software engineering benchmarks provide information on project scope through metrics such as code length and source control.

1.4 Report Outline

The remainder of the report is structured into four chapters. Each of the chapters describe a major component of the work. The remaining sections are organized as follows:

Section 2 outlines the technologies that the *en-co.de* infrastructure is built upon, specifically the QR Code in Section 2.1, the Google App Engine in Section 2.2, and the Bootstrap and JQuery front-end libraries in Section 2.3. The next chapter describes the project design and architecture. This chapter is

broken into three subsections that detail the architecture of *en-co.de* in Section 3.1, optimizing the QR code in Section 3.2, and the approach to software testing design in Section 3.3. Following design and architecture, the results of the project are described qualitatively in Section 4.1 and quantitatively in Section 4.2. To conclude the report, a summary of contributions is provided in Section 1.3, and future work is described in Section 5.1.

Chapter 2

Representative Technologies

This section describes the technologies that form the basis upon which the *en-co.de* web service is built. The technologies were selected based on their quantitative and qualitative characteristics which are described in this chapter.

2.1 Quick Response Code

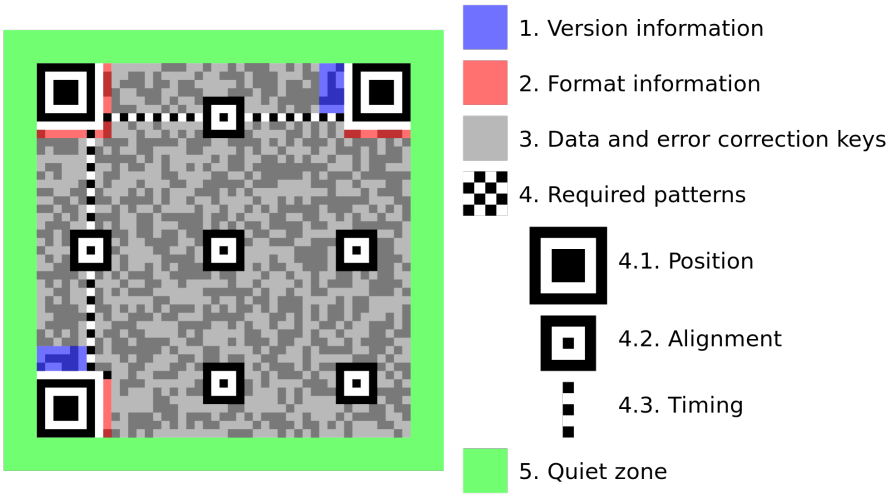

Figure 2.1: QR Code functional layout standards.

The Quick Response Code, or QR Code, is a trademark for a type

of two dimensional bar code that is capable of encoding and storing data. The code is designed so that it is optically readable and encodes information based on a structured pattern of dots or squares. Figure 2.1 shows a typical QR code with additional annotations highlighting the functional layout and structure of the code [10]. The code consists of a two-dimensional patterned grid containing information that allows a camera to properly orient the code for decoding in addition to space required to encode the required information on the tag. QR codes can be created with varying densities that can increase the amount of information that can be decoded on the tag, but requires a higher resolution camera to accurately decode. Figure 2.2 shows two different versions of QR codes, version 1 and version 10 codes, which can encode 20 and 311 alphanumeric characters with M-level error correction (15% of the code can be destroyed and still be read), respectively. The version of the code specifies its density, and therefore its information capacity. The versions increase with increasing density, with the maximum density QR code being a version 40 code with an information capacity of 1264 alphanumeric characters. There are 4 levels of error correction: L, M, Q, and H which allow for 7%, 15%, 25%, and 30% of the codes to be destroyed before the code becomes unreadable, respectively.

The QR code was chosen over other physical encoding mechanisms for several reasons. Compared to RFID, QR codes a lower cost to the user: according to RFID Journal, RFID tags have decreased in cost to 5 cents when bought in volume [7]. While this cost may seem low, QR codes can be printed

Figure 2.2: Version 1 QR code compared to version 10 QR code.

with little to no cost to users of *en-co.de*, with the exception of the cost of paper. However, the most significant factor is the requirement upon the user of the tag: RFID tags requires a specialized scanner to retrieve the information stored in an RFID tag, while a simple camera can be used to decode a QR code. In effect, the reader for QR codes is ubiquitous—most people have camera phones, while few have portable RFID scanners.

2.2 Google App Engine

The Google App Engine [5] serves as the platform upon which the *en-co.de* web service is built. Google's App Engine is a platform as a service (PaaS) cloud computing environment maintained by Google, Inc. for developing and hosting web applications. Applications developed on App Engine automatically scale across servers on heavy load and can take advantage of a rich framework for automating tasks and procedures that would generally

require third party resources. While App Engine supports many server-side languages, *en-co.de* is built with that Java language.

En-co.de uses App Engine on many levels, however the main architecture consists of various servlets, or small Java programs that run on the App Engine, which perform a set of operations on the App Engine's cloud database, the App Engine High Redundancy Datastore (HRD). These servlets, referenced by mapping an HTTP Request to a specific servlet via URL, perform operations such as inserting new tags into the datastore, retrieving information relevant to a specific tag, and access user account settings. A simplified model of how App Engine is integrated within the *en-co.de* web service as shown in Figure 2.3.

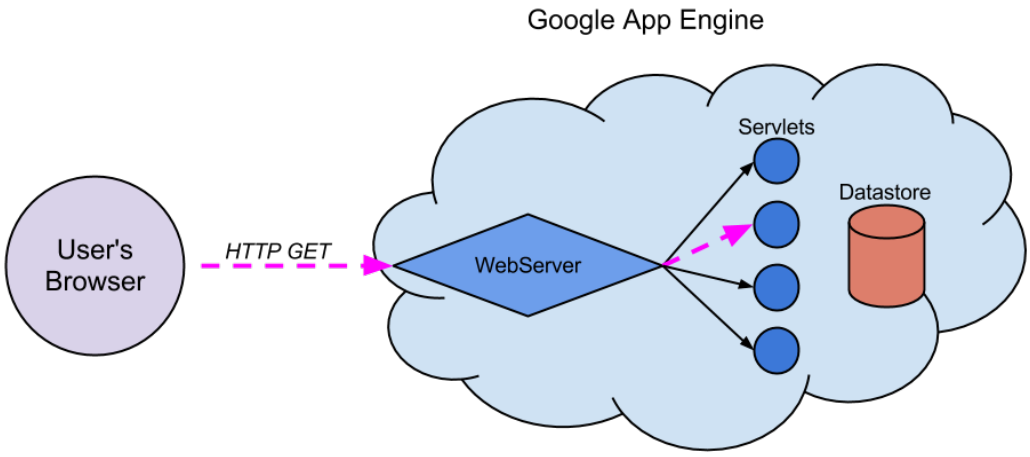

Figure 2.3: Google's App Engine infrastructure diagram.

2.3 Front-end Technologies

En-co.de features a simple and responsive user interface that performs many operations client-side (or within the user's web browser) to reduce the workload on the server. The two primary technologies that the user interface is built upon are the Twitter Bootstrap [6] responsive cascading style sheets (CSS) and the JQuery [11] library for the Javascript client-side scripting language.

Bootstrap provides a comprehensive CSS file that allows users of Bootstrap to quickly and easily build responsive websites. The responsive layouts in Bootstrap allow websites to easily adapt page rendering to many styles of browsers, from smart phone browsers, tablets, or laptops. This included functionality takes much of the burden of development off the web developer. Bootstrap provides, in addition to responsive layout, modern styling and icon packages allow developers to create simple but elegant websites. *En-co.de* uses Bootstrap's styling, icons, and responsive properties to provide a simple and elegant user interface. *En-co.de's* layout is designed using Bootstrap's scaffolding features and the buttons and icons are from Bootstrap's base configurations.

JQuery is a common Javascript library that simplifies the task of querying the document object model (DOM) for elements within a page and performing actions on the selected elements. JQuery is used in *en-co.de*, for example, to dynamically update pages based on server responses to user button clicks without having the user reload the page. In general, each user interface

element on *en-co.de* has an associated servlet which is queried using asynchronous Javascript calls (Ajax) that update the element. This pattern exists across the site and JQuery provides a simplified method for performing these actions. *En-co.de* uses JQuery extensively for a responsive interface: JQuery provides an Ajax API that is used to submit form elements and receive and parse JSON that is returned from the Java Servlets that make up the *en-co.de* server functionality. As an example, JQuery is used to parse JSON returned when querying for a list of tag ID's that belong to a specific user. *En-co.de* then asynchronously populates a table in the HTML content in the page with the list of tag ID's.

Chapter 3

Project Design and Architecture

The *en-co.de* web service is designed for scalability in the face of large amounts of concurrent data requests. In addition, *en-co.de*'s database architecture is designed to minimize the number of read and write operations necessary by keeping the number of indexes in the cloud storage, or HRD, minimal. Beyond scalability, the web service is designed to function asynchronously: page load times are not governed by servlet response times for data queries.

While the web service is designed to be maximally efficient given design constraints, I have also implemented a software testing approach to the site architecture to ensure functionality of the servlets across versions of the software. Access to the HRD and responses to servlet calls are verified with a unit testing suite that can be run without testing the site in production environments. The following sections describe these concepts in more detail.

3.1 Architecture of the Web Service

The *en-co.de* web service architecture can be described in three distinct components: the database architecture, the site architecture, and the

application programming interface.

3.1.1 Database Architecture

The database architecture in *en-co.de* is designed around meta data that is associated with a tag identifier, or tag ID. Each *en-co.de* tag encodes a unique identifier that is used to query meta data associated with the tag stored in the HRD. To facilitate this interaction, the HRD maintains two tables, an *en-co.de* tag table, and an *en-co.de* user table. The tag table associates tag ID's with tag meta data, while the user table associates *en-co.de* user names (or owners) with user meta data and a list of tags belonging to the user. A UML diagram in Figure 3.1 illustrates the database model.

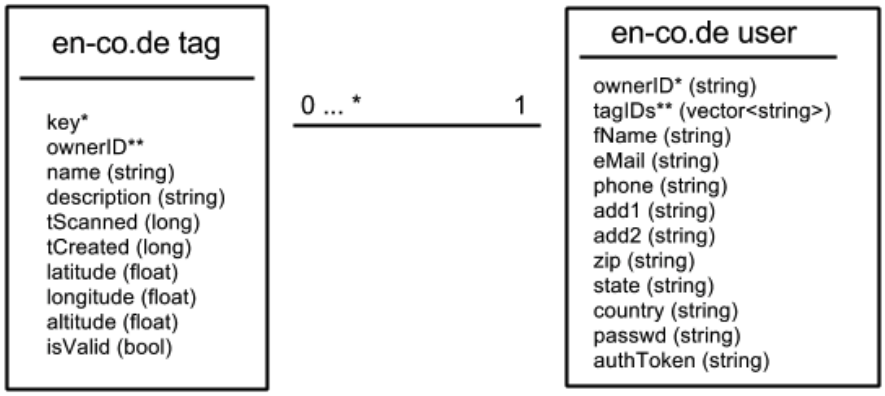

Figure 3.1: UML diagram of database layout.

Most of the data associated with the tag and user is self explanatory. Interesting elements of the tag table include the location properties latitude, longitude, and altitude. These values represent the location of the tag at the

time it is scanned by someone who has found a tag and scanned it. These properties allow the owner of the tag to better track the tag location. The user table contains information that allows the owner to authenticate to the *en-co.de* web service. The password property contains an MD5 hash of the user's password, which is generated in the user's browser. The authentication token is the password hash combined with a secret known only in the *en-co.de* source and further obfuscated with randomized values. The authentication token is given to the user's browser and allows the user to initiate updates to tag and user tables through the site or application programming interface (API), covered in Section 3.1.3.

Within the servlet code, the HRD is queried and updated with meta data objects that reflect the properties associated with a key. There are two such objects in *en-co.de*, UserMetaData and QRMetaData. Each of these objects contain a identifier field that uniquely identifies them within the database (the key), and the other fields corresponding to the UML diagram in Figure 3.1. Using object representations of data allows for simple and less error-prone usage of the HRD: it is possible to manipulate objects directly rather than perform specific queries. This usage methodology is made possible by using the Objectify App Engine library. As an added benefit of using the Objectify library, HRD accesses are cached and future transactions need not necessarily access the HRD, but may simply access the local cache, or *memcache*.

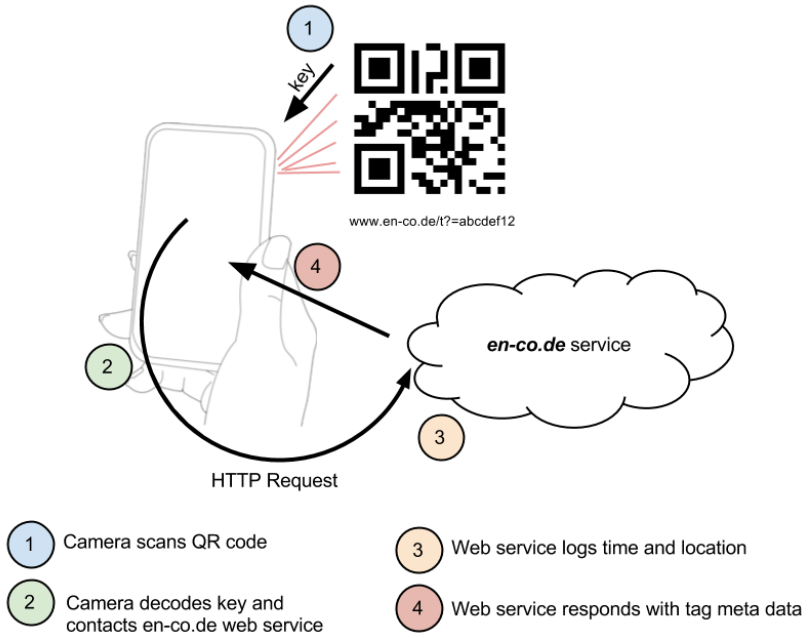

Figure 3.2: Web service usage scenario.

3.1.2 Site Architecture

The *en-co.de* site is built around the concept that someone will scan the QR code associated with a physical object, an action which will redirect the scanner to the tag's unique page and perform actions on the *en-co.de* infrastructure. A typical scenario is illustrated in Figure 3.2. In this scenario, the user scans an *en-co.de* tag. The tag contains a URL that is a combination of the *en-co.de* domain with a query parameter specifying the tag ID. The scanning application redirects to the tag's page through an HTTP request. This request determines the user's location using the HTML5 geolocation API

and forwards the location to an *en-co.de* servlet. The *en-co.de* service responds to the user's HTTP get request with a page populated by the meta data associated with the tag ID. If the owner of the tag allows for email and text notifications, the owner of the object in which the tag is affixed will at this point receive a text message and email informing the owner that one of their objects have been scanned.

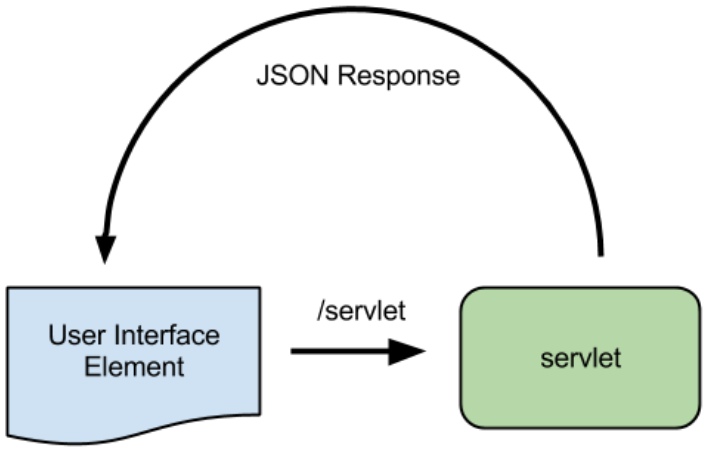

Figure 3.3: Relationship between UI elements and servlets.

The site architecture follows this usage scenario closely from the standpoint of the client who scans the tag. From the user, or tag owner perspective, the site architecture is centered on data management and creation of tag meta data. The site layout allows the user to create individual tags and assign meta data to them individually, or batch create up to 50 tags simultaneously. Batch tags can have their data initialized when they are first scanned, allowing the user to attach tags to more items in a shorter amount of time when compared

to individually creating tags. Following these usage scenarios, the site's architecture is designed so that every action on the interface of the site has a corresponding server-side servlet that performs data management tasks on the HRD. For example, viewing the set of tags that belong to a user in a table has a corresponding servlet that serves an array of tag meta data in JSON format. The site ingests the JSON data returned from the servlet data and dynamically populates the table. In another case, the user may populate his account settings, including his address and phone number, whereupon submitting the form a servlet is called that updates the user's meta data in the HRD. This relationship between user interface elements and server-side servlets is illustrated in Figure 3.3.

3.1.3 Application Programming Interface

En-co.de's design allows for utility beyond the web interface that is accessed through the root domain. *En-co.de* implements a custom API for access to the web service on mobile devices or other clients that may have no need or access to the main site. The API supports the creation, modification, and query of tags through a set of URLs that reply to requests with structured data in the JavaScript Object Notation (JSON) format [8]. This API allows other application developers to use *en-co.de*'s cloud platform for tracking and maintaining tag meta data without having to implement or render a user interface. An example API function is shown below, where a query on a tag ID results in the meta data associated with the tag:

HTTP Request:

```
HTTP GET en-co.de/retrieveTag?key={tagID}
```

JSON Response:

```
{
        "name":"My Dog",
        "description":"Doggy",
        "ownerName":"Joe Simple",
        "email":"email@en-co.de",
        "phone":"555-555-5555",
        "addStreet":"123 Any Street",
        "addCity":"Austin",
        "addState":"TX",
        "addZip":"78704",
        "messaging":true,
        "tracking":true
}
```

3.2 Optimizing the QR Code for *En-co.de*

QR code decoding is achieved using an optical device to capture an image of the code, correct for alignment and orientation using image processing and visual markers, and performing a decoding operation using the QR code

decoding scheme. For this process to work properly, each element in the QR code must fill 3 – 5 pixels in the image sensor. QR codes can vary in density, where the highest density codes can hold the most information, but require a larger and more sophisticated image sensor. In exchange, the code must be scanned at a close distance to maximize the portion of the sensor allotted to reading the code. To optimize the QR codes used in *en-co.de*, a quantitative analysis of QR code optical properties was undertaken.

En-co.de must encode a URL containing the domain of the web service and the unique identifier of the tag, for example a tag could contain the following sequence of characters: `www.en-co.de/?t=abcd1234`. An *en-co.de* tag requires 24 ASCII characters, necessitating a Version 2 code consisting of 25 x 25 elements. Using this version code, we can generate tags with a 30% error correction rate while still encoding all 24 characters. Interestingly, the 8-character identifier can generate 2 trillion unique permutations, allowing for a large amount of unique and active *en-co.de* tags at any one time. The Version 2 tag, proceeded only by the version 1 tag in low density characteristics, is the ideal density by which to encode the URL and identifier: ideally the lowest density tag possible should be used because it increased the range at which a camera device must be from the tag for a successful reading.

In *en-co.de*, it is important to make the tags as small as possible to decrease the frustration users might have to attaching a tag to their belongings. To better understand the minimum size of a tag given a fixed distance, we explore the mathematical relation between the minimum size of a tag versus

the tags encoding density at a fixed distance:

$$\text{size}_{min} = \text{distance} \times tan\,(\text{FOV})$$

Using this relation we can chart the minimum size of a QR code given the encoding density, a fixed scan distance, and a fixed camera resolution. Here, we assume a fixed scanning distance of 6 inches, a camera resolution of 5 mega-pixels, and a camera field of view (FOV) of 60 degrees. These assumptions are based on specifications of the *Apple iPhone 4*, a widely used smart-phone released June 24, 2010. From the relationship shown in Figure 3.4, it can be seen that a Version 2 QR Code's minimum size is approximately one inch in both width and height.

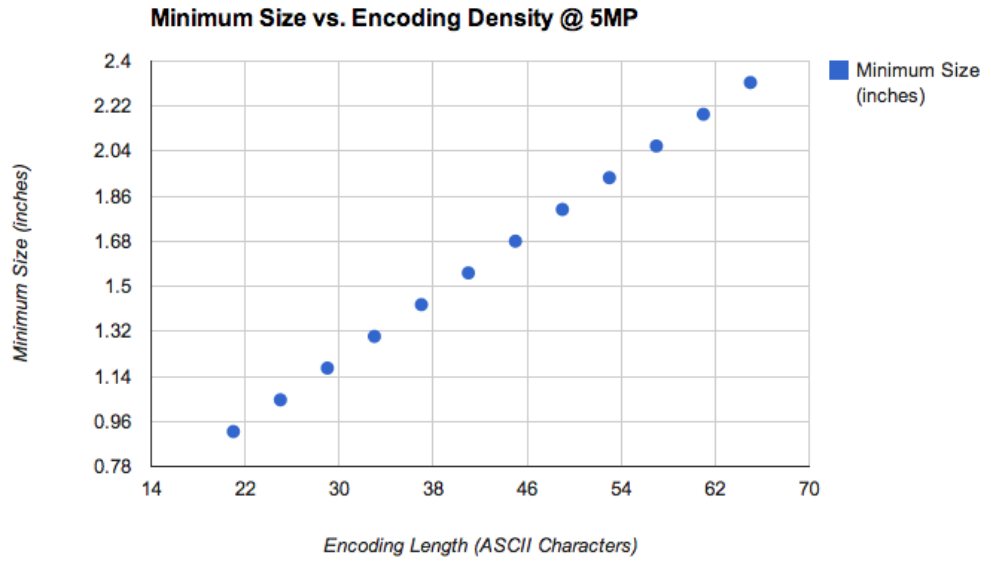

Figure 3.4: Relationship between encoding density and minimum size in QR tags.

3.3 Approach To Software Testing

Testing web applications is challenging because they are generally built around a web server process that serves pages or content. This places a requirement on the testing framework: the web server must be running to properly evaluate the server functions of the web service. While it is feasible to run a local instance of the web server to test basic functionality of the web service and to ensure proper responses from servlets, the problem becomes problematic in unique environments that rely heavily on cloud platforms, such as the Google App Engine because of the variation in response time and data coherency in production environments in comparison to development environments.

3.3.1 Testing with the High Redundancy Datastore

App Engine's HRD can be instantiated locally as part of an App Engine local process to facilitate test, but its behavior may vary from the behavior of the HRD in production. This variance between production and emulation is because the HRD does not guarantee absolute consistency of the datastore at any given time: if a user writes an entry into the datastore, a query for the updated element may return stale results. This surprising result is a consequence of the distributed nature of the datastore. The HRD exists across many geographically separated data centers, and elements within the datastore can only be guaranteed to be eventually consistent.

Eventual consistency can result in anomalous behavior at the user interface: for instance the user may create a new *en-co.de* tag, but not see the

tag reflected in their tag database because the query for the user's tags does not necessarily return the latest state of the datastore. This behavior can be overcome by properly designing the data formats such that elements that share some common connection should be grouped according to that connection. This grouping is defined using a common ancestor, or parent key. In *en-co.de*, each tag has a parent: the user name of the owner of the tag. The owner of the tag is guaranteed to see the latest updates to tags that belong to him because of the tags he/she owns share an ancestor and are therefore guaranteed to be consistent.

Testing may seem difficult in this situation: how can we properly evaluate the correct functionality without testing on a live version of the web application? Google provides as part of its App Engine a testing API that contains a local (in memory) version of the datastore that can be instantiated without running a copy of the App Engine as a whole. This allows the design of unit tests using JUnit that can instantiate a local copy of the datastore and verify behavior. In addition, it is possible to simulate worst-case eventual consistency by setting the job policy in the local datastore to maximally unapplied: any write to the datastore will not be reflected by a query that does not specify the ancestor of the written object. *En-co.de*'s test methodology uses a maximally inefficient in-memory datastore to verify correct behavior of its database access routines. Specifically, it tests the addition of elements in the datastore, querying for their existence or field values, and the proper deletion of datastore elements.

3.3.2 Testing Servlet Response

Testing servlet response requires a running version of the web service, either locally or in production. *En-co.de*'s testing infrastructure makes use of HttpUnit [4]. HttpUnit emulates browser behavior with minimal code and allows for the verification of expected responses from servlets. *En-co.de* uses HttpUnit in conjunction with JUnit to evaluate the JSON responses of the servlets involved in accessing and managing the tag database. This simple process verifies the expected behavior of servlets, and can be run as a separate JUnit test suite.

Chapter 4

Project Results

The results of the *en-co.de* web service can be expressed qualitatively and quantitatively. The qualitative results are expressed in the form of images of the web service interface design and responsiveness or site structure. Qualitative results center on the responsiveness of the service, cost analysis of the datastore, and software engineering benchmarks.

4.1 Qualitative Results

The qualitative results of the project are expressed through images of the *en-co.de* interface which give an overview of the entire structure of the web service.

4.1.1 Interface

The *en-co.de* Internet interface is designed to be simplistic and feature complete to design specifications. The interface allows users to create a new *en-co.de* account, create new tags individually, create tags in batch operations, view created tags while sorting by properties, and receive alerts on scanned tags. Finally, the interface allows a scanned tag's meta data to be viewed on

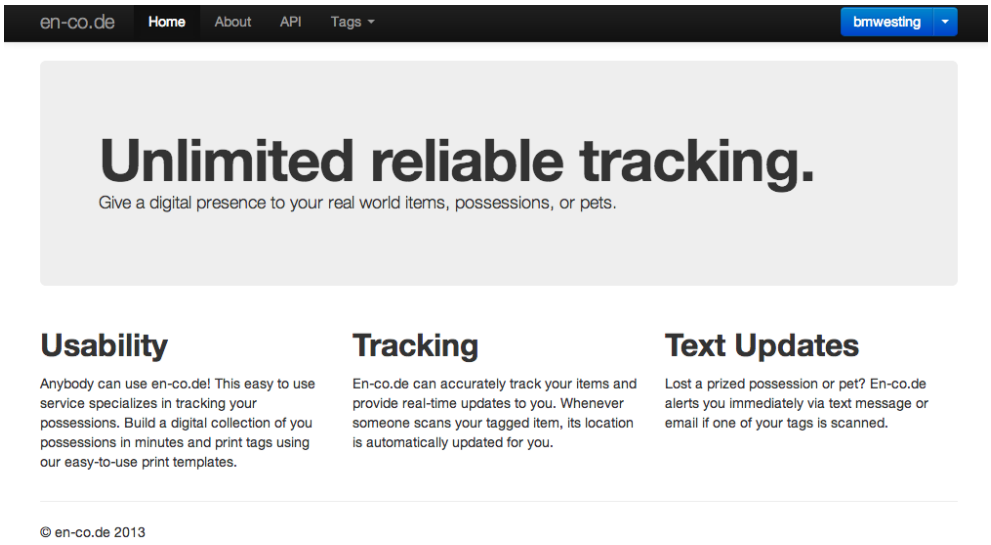

Figure 4.1: The home page of the *en-co.de* web service.

a portable device.

Figure 4.1 shows the home page, or what users see when they navigate to the *en-co.de* URL. The *en-co.de* home page explains the features of service and provides a navigation toolbar at top that can be used to login to the service or navigate to different parts of the website.

The user can update their personal information and elect which information to share in the account settings page shown in Figure 4.2. By default, users share no personal data, but may choose to do so from this page. The account settings page also contains the token code used for API requests initiated by the user.

When creating a new tag, the user can specify the name and description of the item. A QR code is automatically generated for the user. The data

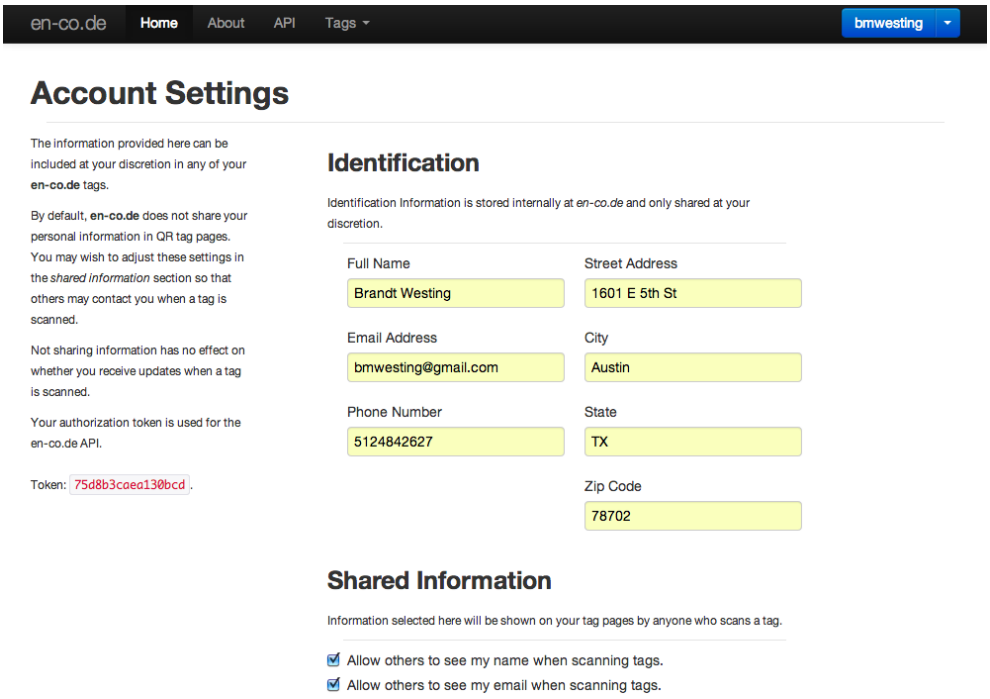

Figure 4.2: The account settings page of the *en-co.de* website.

entered here can be edited at any time, and is shown when another user scans the QR code and is directed to the tag's page. Figure 4.3 shows the interface for creating a new tag.

The *View Tags Page* in Figure 4.4 allows the user to quickly view and sort the collection of tags belonging to the user. In addition, the user can quickly ascertain when a tag was last scanned and the location of the scan.

When a tag is scanned that belongs to a user of *en-co.de*, the location of the tag is collected from the scanner's browser. The user can then accurately track the position of the tag, as shown in Figure 4.5.

Figure 4.6 shows the interface of *en-co.de* when viewed on a mobile

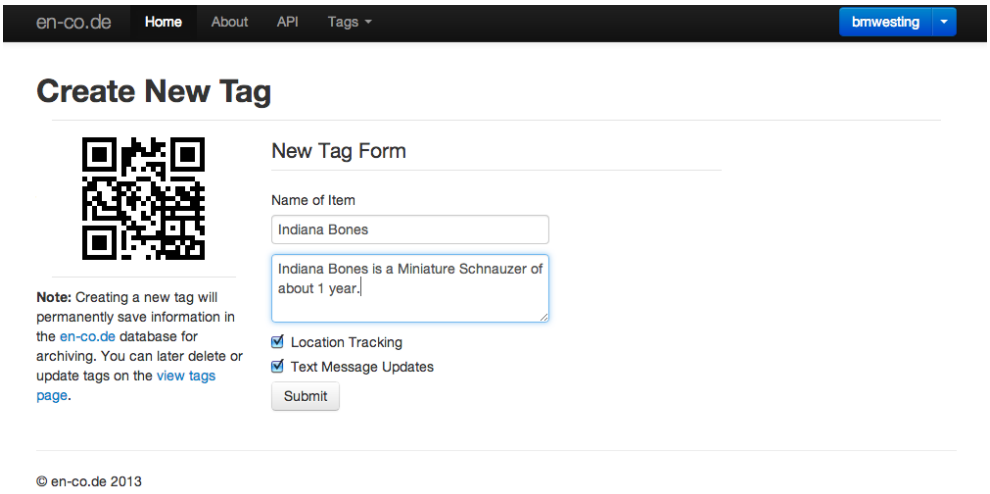

Figure 4.3: New tags can be created individually or in batch.

device. When someone scans the QR code, the software on the scanning device redirects the user to the tag's page (the URL of which is encoded in the tag). Upon reaching the tag's page, the users is asked permission for access to their location. If accepted, the location of the scanning device is logged in the *en-co.de* database so that the owner of the tag can track the scan location. If the scanning device denies permission to use location information, the location is estimated using geographic IP (GeoIP) location information. GeoIP is generally only accurate to 15 miles of the scanned location, but can occasionally be incorrect by a more significant margin. Accurate location tracking is facilitated with the HTML5 GeoLocation API. When used successfully, the API will query the user's hardware for GPS information as appropriate and is as precise as the hardware's GPS can report. In contrast, GeoIP uses the user's IP address as a key for an online geolocation database to determine the

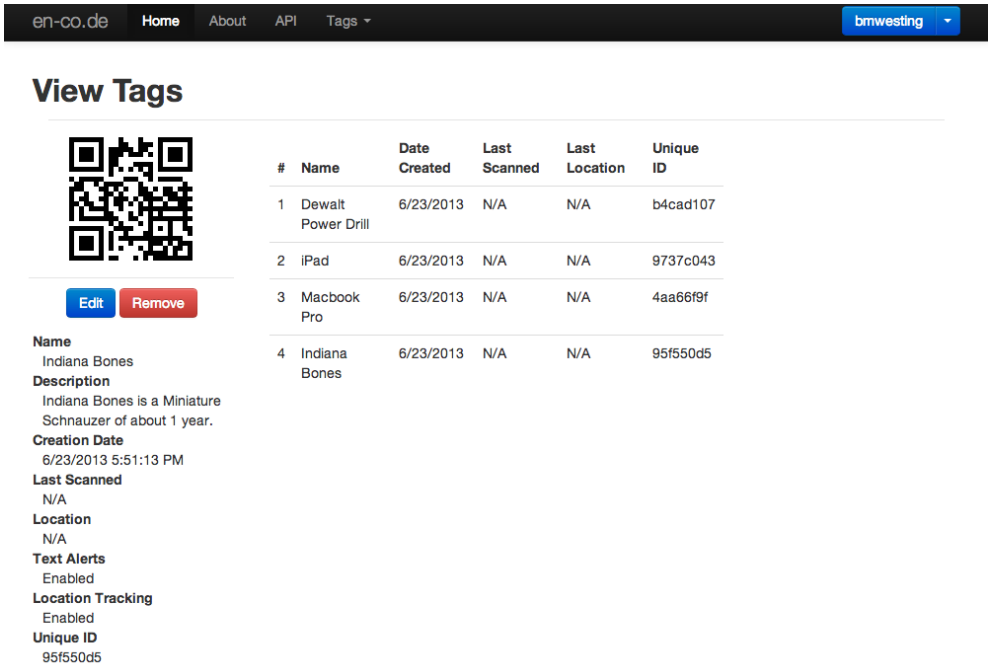

Figure 4.4: The View Tags Page in *en-co.de*.

user's approximate location. Geolocation databases, however, provide limited location accuracy, are constantly changing, and subject to availability of the geolocation database provider.

Once the tag is scanned, the owner of the tag is sent a notification email and a notification text message only if they elected for tracking and messaging notifications when creating or last editing the tag.

4.2 Quantitative Results

The quantitative results of the project are broken into three subsections: the response times and metrics involved in the web service (Section 4.2.1), a

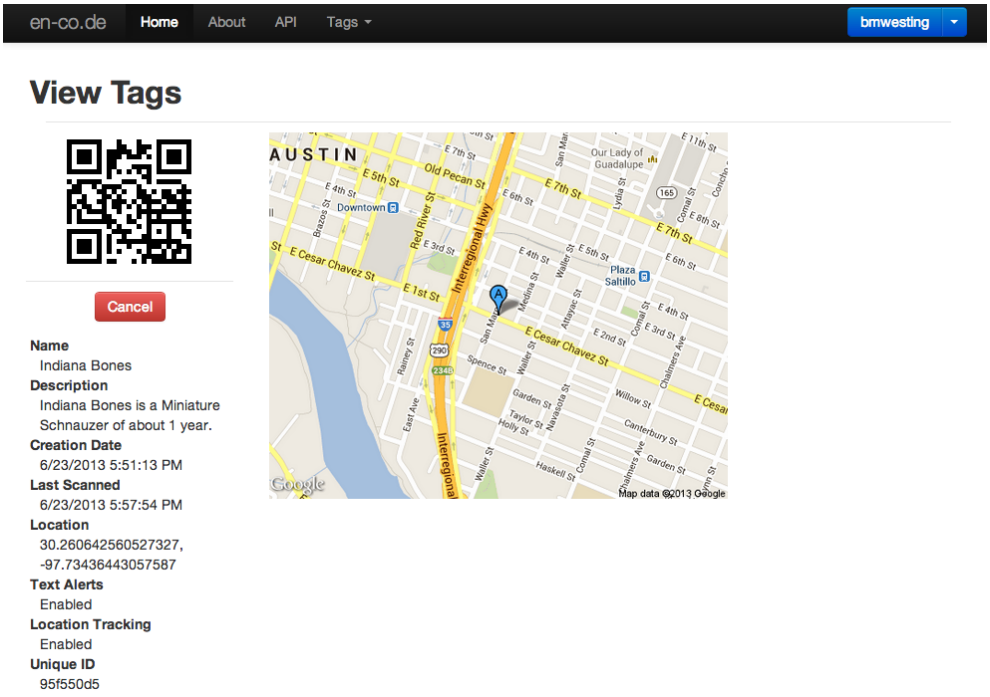

Figure 4.5: The location of the tag when scanned is shown.

cost analysis that explores the cost of operating the service, specifically the datastore (Section 4.2.2), and software engineering metrics such as lines of code in the project, version control metrics, and bug metrics (Section 4.2.3).

4.2.1 Response Metrics

Website responsiveness is measured by averaging timing results from page loads using a third-party client (pingdom.com). Measuring response times in this way eliminates the possibility of local latencies on the measuring device, such as residential connection latencies or bandwidth issues. *En-co.de* runs on top of a scalable cloud infrastructure provided by Google. At the

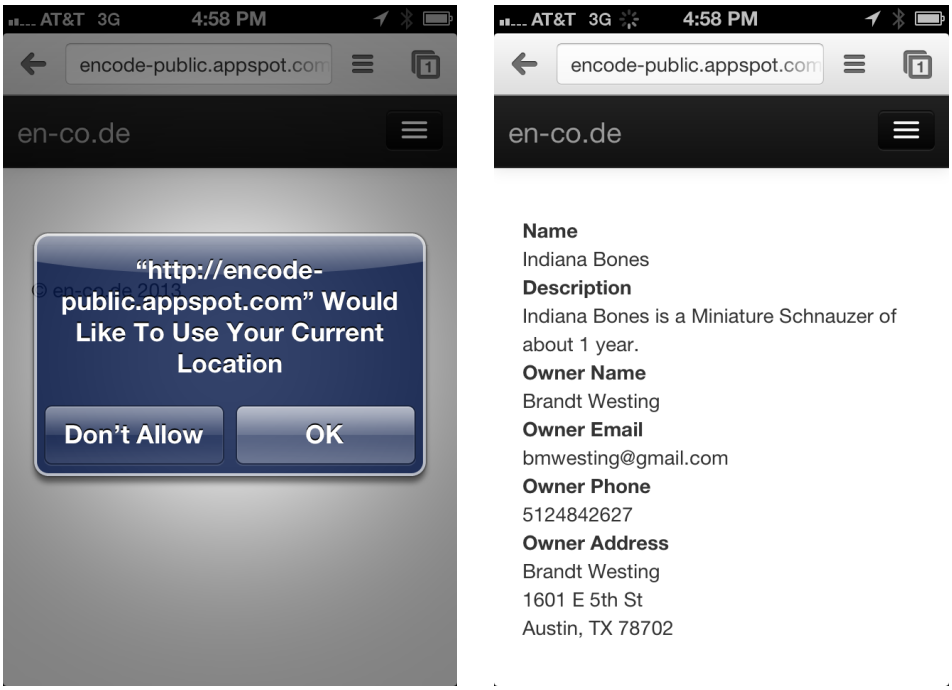

Figure 4.6: Scanning a tag prompts the user for location permission.

time of the tests, *en-co.de* was running on a 600 MHz F1-class instance using 128 megabytes of RAM to serve pages. The *en-co.de* cloud configuration was specified for automatic control of *Max Idle Instances*, or how many instances of the server run simultaneously to serve pages to users. Automatic control of *Max Idle Instances* results in minimal instances running, and therefore lower costs during low usage time. However, when the number of instances can no longer handle web traffic, new instances must be started resulting in a high startup latency, or warm-up time, for those users waiting on the new instance. A warm-up time of 4.7 seconds on average was measured by forcing new instances to be loaded and their response times measured and averaged.

Average response times of a page load were 1.2 seconds when averaged over 10 page loads of the *en-co.de* home page. Latency, as reported by App Engine and averaged per hour, showed 63 milliseconds response latency. In general, observed latency was less than 100ms, but transfer times for page resources (such as Javascript and CSS files) accounted for the longest wait times with some taking up to two seconds to be loaded. Page reloads resulted in large reductions of over a magnitude in page load time due to caching of page elements such as Javascript and CSS resources.

4.2.2 Cost Analysis

Analysis of costs associated with operating the *en-co.de* infrastructure is centered on two primary cost factors: the cost of serving pages and the cost of database access. Google's App Engine infrastructure is funded by charging

users for services. App Engine charges for server uptime in the form of instance costs, datastore storage costs per month, and datastore access costs per month.

Frontend Class	Memory Limit	CPU Limit	Cost Per Hour Per Instance
F1	128MB	600MHz	$0.08
F2	256MB	1.2 GHz	$0.16
F4	512MB	2.4 GHz	$0.32
F4_1G	1024MB	2.4 GHz	$0.48

Table 4.1: Instance Costs with Google App Engine

App Engine, by default, can serve pages for free when traffic is low. App Engine provides 28 instance hours of usage for free per day. *En-co.de* runs on one 600 MHz instance during low usage. It is therefore free to host *en-co.de* when only one instance is necessary to serve pages. Beyond the single instance, App Engine charges for service hours and instance type. When page requests outstrip the ability of the default instance to handle requests, another instance is automatically started that begins serving pages. The associated costs of new instance are shown in Table 4.1. *En-co.de* is configured to use F1 default instances.

The single highest cost element in the *en-co.de* service is datastore access. Datastore storage is $0.18 per month per gigabyte of storage. The single most common element in the datastore is the QRMetaData element (Section 3.1.1), whose average size can be approximated to 512 bytes over all tags per tag. Therefore, 2 million tags would cost approximately $0.18 per month. Given an average number of tags per user being 100, and the number of users being 10 million, the monthly cost of data storage would be $92.16.

Service	Cost Per Unit	No. Monthly Units	Cost
Storage	$0.18/GB	512	$92.16
Datastrore Read	$0.06/100k	780 Million	$468.00
Datastrore Write	$0.09/100k	18.7 Billion	$16,830.00
		Total Cost	$17,390.16

Table 4.2: Monthly Costs Associated with Datastore

App Engine operates on a monetary charge for datastore read and write operations. App Engine grants 50,000 free read and write operations on the datastore per month. Beyond these free reads and writes, Google charges $0.09 per 100,000 write operations and $0.06 per 100,000 read operations. Given there are approximately 100,000 seconds in a day, an average of 300 (one tag scan per month per item) QR tags are read per second, and 300 QR tags are written per second, the cost per month of datastore access can be estimated. Reading a tag from the datastore amounts to a single read operation, and given 26 million read operations a day, the monthly cost can be estimated at $468.00 for datastore reads associated with tags. Writing a single QR Tag meta data object requires 24 write operations due to the indexes and number of field updates. Assuming 26 million tag writes per day, with 24 datastore writes per tag, we can estimate a monthly cost of $16,830.00.

Expected monthly costs given 10,000,000 users with 100 tags per user, and a tag read and write every second, datastore costs is summarized in Table 4.2.

4.2.3 Software Engineering Metrics

Software engineering metrics involve the number of lines of code in the *en-co.de* web service, version control metrics, and other metrics. These metrics help show the effort involved in creating a similar project and explain the scope of work involved in creating a web service like *en-co.de*.

En-co.de consists of Java-language servlets for back-end services such as datastore access, authentication, and data analysis. *En-co.de*'s front-end is composed of Java Servlet Pages (.jsp) files written in HTML and Javascript. There are 17 Java servlets, and 7 Java utility classes that the servlets use for datastore access or authentication, comprising 1102 lines of Java code. The web interface consists of 11 JSP files that are comprise of 1421 lines of HTML and JavaScript in aggregate, of which approximately 60% is JavaScript. There are In addition to the JSP and Java, there are 180 lines of assorted XML in 3 XML files used to describe the web service behavior, identify datastore indexes, and match URLs to servlets. These results were measured using the CLOC utility. Code was committed to a Git repository hosted by BitBucket in 38 commits spanning from March 13th, 2013 to June 21st, 2013.

Bugs were present and tracked in many parts of the *en-co.de* service in both client and server-side software. Logged bugs in client-side code were related to JavaScript execution, particularly in the *en-co.de* secure sign-in service. Secure sign-on functionality was implemented using local storage, MD5 hashing, and Ajax. The most difficult bug related to secure sign-on involved the ability for the service to remember a user after they navigate away from the

site and return. Due to the architecture of the secure sign-on implementation, if the user visited the site on more than one device, for example a laptop and a tablet computer, the site was unable to authenticate both browsers automatically when navigating to the site. This bug was resolved by using multiple authentication tokens per user. In general, best practice is to use existing secure sign in technologies such as FaceBook or Google's single sign-in services. Server-side bugs are generally easier to avoid and debug because of sophisticated debugging and unit testing. However, performance related bugs are difficult to detect in some cases, and can be costly when involving the datastore. The most costly bug involving *en-co.de*'s datastore usage was related to datastore indexing. By default, the Google App Engine Plug-in for Eclipse automatically generates datastore indexes for the user when datastore query code is present. Because an index on a datastore property doubles the number of writes necessary when making a datastore write on an entity, indexes can quickly cause the cost of operating the site to rise. This performance bug was fixed by disabling the auto-generation of datastore indexes in the App Engine project and manually building the index file.

Chapter 5

Conclusion

5.1 Future Work

While *en-co.de* provides a usable service and clean interface to managing a digital presence for physical objects, its current state is still considered a prototype or proof-of-concept. Currently, there is no model in place for monetizing the *en-co.de* service. Monetizing the service is important for the long-term survival of the project due to the requirement to recoup costs associated with maintaining the web infrastructure, as described in Section 4.2.2. Beyond monetization, *en-co.de* could benefit from rigorous quality assurance and testing to provide a higher quality user experience. The choice of physical encoding mechanism could be tweaked for *en-co.de* to improve the brand image and increase interest from users who might not find the QR code an appealing object to add attach to their possessions. Variations of the QR code could be used to increase the attractiveness of the service. Similarly, user studies could be performed to better understand the market and the acceptance of QR codes in the populations of various countries regarding the general awareness of QR technology among the populace. For instance, it is known at this time that QR codes are widely accepted in Japan, but have less penetration in western countries.

Beyond the areas in which *en-co.de* can be improved, further investigation is needed into areas in which the service can be useful beyond personal tracking tasks. For instance, there are many opportunities for 2D barcodes (such as QR codes) to be used in places where conventional 1D barcodes are used, such as supermarket checkout scanners or the freight industry. The application of a service such as *en-co.de* in these industries may allow customization beyond their current use, such as tracking and categorizing grocery store purchases for an individual. Beyond researching how *en-co.de* can be used in these industries and applications, future work in implementing the methodology behind *en-co.de*, but with alternative tagging technologies such as RFID, could be explored. While RFID is more expensive than QR technology, there may be applications where RFID is more applicable as the encoding mechanism for the web URL and key and where *en-co.de* could be used for tracking, alerts, or data analysis.

5.2 Lessons Learned in Creating *en-co.de*

En-co.de is a web service that spans several programming languages and builds upon a stack of technologies that are the result of decades of advances in Internet technologies. In building *en-co.de*, I learned many lessons. Below I summarize the lessons I learned with a list of five things I would recommend a developer do and not do when starting a project like *en-co.de*.

5.2.1 Five Things To Do:

1. Have a clear understanding of the architecture and technology stack involved in the project before coding.

2. Create a prototype or minimum viable product that achieves the goals of the final product but is not refined before working on details.

3. Use source control from the very beginning of the project and take advantage of tools such as bug tracking.

4. Explain your project to others at a high and low level—it will help you understand the product better yourself.

5. Remember that sometimes less is more.

5.2.2 Five Things Not To Do:

1. Do not re-invent the wheel unless as a learning exercise.

2. Do not focus on interface elements before core functionality is in place.

3. Do not get blocked: ask for help if you are blocked and have exhausted self-help resources to avoid wasting time.

4. Do not work on a project you are not passionate about—it is difficult to excel at something without passion.

5. Do not develop a project without spending equal time testing the project. Fluidity, performance, and correctness can be as important as functionality.

5.3 Brief Survey of Similar Work

En-co.de is similar to other projects in the technologies it uses, but is unique in the service it provides. Augmenting physical objects with digital identifiers and corresponding web resources is present today in many products, but are usually targeted as specific markets. *En-co.de*, however, is general purpose and not targeted at any specific use case. *PetQRTags* [1] use QR codes on pet collars, similar to the user story in Section 1.1.2. These tags offer similar abilities as *en-co.de* tags in that up-to-date user information can be obtained from a *PetQRTag*. However, *PetQRTags* do not support location tracking or email and text message updates. Furthermore, *PetQRTags* must be obtained as part of pet collar and can not easily be printed or used for other purposes beyond pet collars.

The ability to encode a digital message in an image signature is possible with many technologies. Some commercial entities use two-dimensional bar codes for tracking of cataloging of physical objects. FedEx, for example, uses a PDF-417 bar code [3] to encode information into packages. The bar code, however, does not link to digital content stored on a server or in the cloud. Instead, FedEx's tags store information directly in the bar code. Information such as tracking numbers, zip codes, and addresses are stored in the code

in a fixed format. Unlike the PDF-417 barcode, *en-co.de* tags store a single piece of information: a uniform resource locater (URL) that uniquely identifies the code. Data is then retrieved from the URL using the Hypertext Transfer Protocol (HTTP). *En-co.de* tags, therefore, do not require proprietary software for tag decoding and any camera device such as a mobile phone or tablet can access the tags and update the information stored in association with the tag.

Linking an encoded unique identifier to a cloud-based datastore is possible with Radio frequency identification (RFID), and can be made to function in a role similar to *en-co.de* tags [2]. RFID can be used to wirelessly identify an object through the transfer of data between an embedded RFID tag in an object and an RFID scanning device. Upon receiving the identifier present in the RFID tag, a device could query a web service similar to *en-co.de*. RFID scanners are not ubiquitous on portable devices: while cameras can be used to scan a QR code optically and are present on most smart phones today [9], RFID scanners are not present on mobile phones or tablets. In addition, because the RFID tag must be manufactured, their cost is higher than a printable QR code.

More detailed comparisons of similar work is found in Section 1.2, as *en-co.de* relates to similar services, and Section 2.1, where similar encoding technology such as RFID is discussed.

Bibliography

[1] Petqrtag. http://petqrtag.com/, 2011-2013.

[2] Mario Cardullo and William Parks. Transponder apparatus and system. US Patent 3713148, May 1970.

[3] FedEx. *Barcode and Label Layout Specification*. FedEx, January 2004.

[4] Russell Gold. Httpunit. http://httpunit.sourceforge.net/, 2008.

[5] Google Inc. Google app engine. https://developers.google.com/appengine/, April 2008.

[6] Twitter Inc. Bootstrap. http://twitter.github.io/bootstrap/, August 2011.

[7] RFID Journal. Can i buy a 5-cent rfid tag? http://www.rfidjournal.com/faq/show?84, 2002-2013.

[8] Json. Introducing json - rfc 4627. http://www.json.org/, July 2006.

[9] Cellular News. Imaging phones dominate emea shipments. http://www.cellular-news.com/story/11357.php, December 2004.

[10] Internation Standards Organization. Iso/iec 18004:2006 qr code 2005 bar code symbology specification, 12 2011.

[11] JQuery Team. Jquery. http://jquery.com/, August 2006.

Vita

Brandt Michael Westing attended the University of Texas at Austin where he obtained a Bachelor of Science in Electrical Engineering in 2008. He joined the Texas Advanced Computing Center in 2008 as a research engineer where he worked in the areas of scientific visualization and human-computer interaction. He has since joined Microsoft in Redmond, WA as a software design engineer working with the Perceptive Pixel Team on human-computer interaction related technologies.

Permanent address: westing@utexas.edu
1601 E 5th St.
Austin, Texas 78702

This report was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.